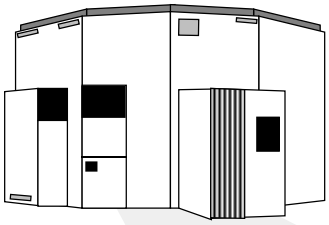
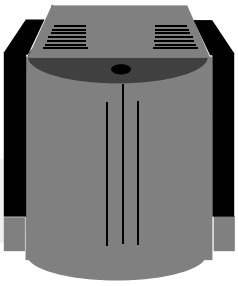
Naval Research Laboratory

Washington, DC 20375-5320





Programming in Perl

Course Notes

Instructor: Michael G. Vonk
Center for Computational Science
(202)767-3884
michael.vonk@nrl.navy.mil

Programming in Perl

1. Introduction

Perl is a public domain scripting language that can be used to automate system administration tasks, to handle Web form results and build Web pages on the fly, and as a general purpose tool. This class serves as an introduction to Perl and covers the following topics:

- script format and execution
- data types
- operators
- control structures
- input and output
- subroutines

Perl is designed to perform tasks too complex for shell scripts, but without the complications associated with languages like C. It is available on a wide range of systems, including UNIX workstations, Macintoshes, and PCs.

2. Perl Script Format and Execution

The following is the ubiquitous "hello world" program written as a Perl script:

```
#!/usr/bin/perl
#
# Description:  Basic perl script
#
# Usage:  hello

print "Hello, world...\n";
```

Example 1 Hello World

Perl scripts are executed (under UNIX) by first granting execute access to the script file and then naming it on the command line (assuming the current directory is set in your path variable):

```
% chmod u+x hello
% hello
```

Perl scripts are completely parsed and compiled before execution, making them fast once execution starts, and eliminating syntax errors once execution begins.

Notes: All statements end in semicolons.

Comments begin with a pound sign and continue until the end of the line.

3. Data Types

The following types of variables are available:

- scalars
- arrays of scalars
- associative arrays of scalars

Perl's name space is exclusive—you can use the same name for a scalar variable, an array, an associative array, etc. without any conflicts. Perl determines data type based on context.

3.1. Scalars

Scalar variables can be any of the following:

- integers
- floating point numbers
- strings

Perl variables do not need to be declared. Undefined numeric variables default to zero—undefined string variables default to the null string. Scalar variables are *always* referenced using the '$' prefix.

Numeric examples:

```
$a = 12;
$b = $a * 3.14159;
$c = 4.06E2;              # Scientific notation
$d = 0177;                # Octal—starts with '0'
$e = 0xFF;                # Hex—starts with '0x'
```

Strings can be delimited by quotation marks, apostrophes, or backquotes. Depending on the delimiter used, various types of substitutions within the string are performed:

"*string*"	Variable substitution and backslash escaping
'*string*'	Only \' and \\ escapes are interpreted
`*command*`	Command output

String examples:

```
$class = "Programming in Perl";
$name = `whoami`;
print "Hello $name, welcome to $class\n";
print 'The value of $name is not given here';
```

3.2. Arrays of Scalars

Arrays are ordered lists of scalar data and are referenced using the '@' prefix. Array assignment occurs as follows:

```
@a = (406, 714, 511);
@b = ("Hello", "world");
@c = (0..9);                # list constructor
($d, $e) = (1, 2);          # same as $d=1; $e=2;
```

Array elements are referenced by their numeric index:

```
$a[1] = 755;
($f, $g) = @c[5,6];         # array slice (note the '@')
```

Array indexes begin at zero by default. The base index for **all** arrays can be changed by resetting the `$[` variable.

The last valid index of an array is given by `$#`*`array-name`*. The size of an array is thus:

```
$size = $#array-name - $[ + 1;
```

Assigning a value beyond the end of an array extends it.

Perl has built in operators to add or remove elements on the right side of an array (`push` and `pop`), or on the left side (`shift` and `unshift`), and to reverse or sort arrays (`reverse` and `sort`).

3.3. Associative Arrays of Scalars

Associative arrays are unordered lists of 'key-value' pairs and are referenced using the '%' prefix. Keys may be any scalar value, including non-integers and negative numbers. Associative array assignment occurs as follows:

```
%budget = ("food", 100,
           "entertainment", 50,
           "lodging", 150);
```

Associative array elements are referenced by key:

```
$budget{"food"} = 125;
```

Perl has built in associative array operators to delete 'key-value' pairs (`delete`), to return a list of keys or values (`keys` and `values`), and to loop through all 'key-value' pairs.

3.4. Scalar vs. Array Context

If an operator expects a scalar operand, it is said that the operand is being evaluated in a *scalar context*. Likewise, operators expecting array operands are said to evaluate those operands in *array context*.

Sometimes this can lead to unexpected results. For example:

```
@a = (406, 755, 511);
$a = (406, 755, 511);
```

The first line creates a new array `@a` from a list of initial values. The second line sets a scalar variable `$a` to the length of an array.

Taking advantage of array vs. scalar context, array size can be obtained simply by using:

```
$size = @array_name;
```

Other operators, like the filehandle operator (discussed later), yield different results depending on whether they are used in scalar context (reads one line), or array context (reads all lines).

4. Operators

Expression Operators

String	Numeric
. (concatenation)	+
x (string repetition)	-
	*
	/ (floating point)
	** (exponentiation)
	% (modulus, integer)

Comparison Operators

String	Numeric
eq	==
ne	!=
lt	<
le	<=
gt	>
ge	>=

Note: Don't confuse string and numeric comparison operators—they are roughly the opposite of Bourne shell `test` operators.

Boolean Operators

&& (and)
|| (or)
! (not)

Matching Operator

`$string =~ /regex/` true if `$string` contains *regex*
(see examples in next section)

Binary Assignment Operator

`$a op= $b;` shorthand for `$a = $a op $b;`

Autoincrement and Autodecrement

```
$a++   $a--
++$a   --$a
```

Precedence

See precedence table. Use parantheses to override.

5. Control Structures

Perl's conditional and looping constructs are similar to those in other languages. The condition tested is either true or false as shown in the following "truth table":

	true	**false**
numeric	non-zero	0
string	non-empty string*	empty string*
array	non-empty array	empty array

* A string containing a single character "0" is considered false.

5.1. Conditional Statements

Syntax for the if-then-else construct is as follows:

```
if (condition) {
    statement-block
}
elsif (condition) {
    statement-block
}
else {
    statement-block
}
```

Another conditional construct is `unless`:

```
unless (condition) {
    statement-block
}
```

5.2. Looping Statements

The `while` and `until` constructs have the following syntax:

```
while (condition) {
    statement-block
}
until (condition) {
    statement-block
}
```

The `for` and `foreach` statements are interchangeable. They come in two forms:

```
for (initial; condition; increment) {
    statement-block
}
foreach value (list) {
    statement-block
}
```

Examples:

```
for ($i=0; $i <= 10; $i++) {
    print "$i ";
}
foreach $key (keys %array) {
    print "at $key we have $array{$key}\n";
}
```

Perl has two constructs for jumping out of loops:

`next`	jump to next iteration of loop
`last`	break out of loop

5.3. Regular Expression Matching

You can search for patterns within strings using the pattern matching operator:

```
$string = "Perl programming";
if ($string =~ /Perl/) {
    ...
}
```

The following regular expressions can also be used:

`.`	matches any character (except newline)
`[abc]`	matches any one of the characters specified
`[^abc]`	matches any one of the characters not specified
`^`	matches beginning of string
`$`	matches end of string
`+`	matches preceding pattern one or more times
`?`	matches preceding pattern zero or one times
`*`	matches preceding pattern zero or more times
`(...)`	groups pattern into single element
`(...|...)`	matches any one of the patterns specified

Examples:

`/^$/`	empty string
`/(Perl|CGI)/`	Perl or CGI
`/80[23]?86/`	8086, 80286, 80386

Additional regular expression characters can be used to match non-alphanumeric entities:

`\w`	matches any alphanumeric character (plus "_")
`\W`	matches any non-alphanumeric character
`\b`	matches word boundaries
`\B`	matches non-word boundaries
`\s`	matches whitespace
`\S`	matches non-whitespace
`\d`	matches numeric
`\D`	matches non-numeric

Examples:

`/^\s*$/`	matches string containing only spaces, tabs, and newline characters or an empty string
`/d+/`	matches a string of digits

Using parantheses for grouping, the matched substring can accessed via variables `$1`, `$2`, etc.:

```
$string = "Perl programming";
if ($string =~ /(Perl|CGI) programming/) {
    $language = $1;
}
```

6. Input/Output

Files are read and written via filehandles. Filehandles, written in uppercase, are opened and closed as follows:

```
open(FILEHANDLE,"filename");
close(FILEHANDLE);
```

There are three predefined filehandles—STDIN, STDOUT, and STDERR.

The following modifiers can be used with open:

"<*filename*"	open *filename* for reading (default)
">*filename*	open *filename* for writing (if *filename* exists, it is overwritten)
">>*filename*"	open *filename* for appending
"|*command*"	write output to *command*
"*command*|"	read input from *command*

File status checking tests include:

-e	file or directory exists
-f	entry is a plain file
-d	entry is a directory
-r	file or directory is readable
-w	file or directory is writable
-x	file or directory is executable

6.1. Reading Input

Lines of input are read by evaluating *<FILEHANDLE>*. The following example reads an entire file, echoing each line to standard output:

```
open(INPUT,"input.dat");
while ($line = <INPUT>) {
    print $line;
}
close(INPUT);
```

In an array context, entire files can be read in at one time:

```
@records = <DATA>;
```

6.2. Writing Output

Two output commands are `print` and the C style `printf`:

```
print "Hello, world...\n";
printf("Sum = %d\n",$a+$b);
```

6.3. Formatted Reports

Format statements can be used to format output, including page headers and footers, page length, and justification. Formats can be placed anywhere in the program and are defined as follows:

```
format format-name =
  template
    fieldlines possibly containing
        fieldholders
    valuelines
.
```

For each field line containing a fieldholder, a value line must follow containing a list of scalar variables, one per fieldholder, separated by commas.

The `write` command outputs text to a filehandle using the current format for the filehandle (by default, the format with the same name):

```
write filehandle;
```

Fieldholders

Most start with '@'. Number of characters, including the '@', determine field width.

@<<<<<	Left justified string
@>>>>>	Right justified string
@|||||	Center justified string
@####.##	Fixed precision numeric field

Strings are padded with blanks or truncated as necessary.

Multiline fields use '@*' on line by itself. Filled fields are used to create filled paragraphs, breaking the text into conveniently sized lines at word boundaries, wrapping the lines as necessary.

```
^<<<<<<
```

Corresponding value for a filled field must be a scalar variable containing text, not an expression.

7. Miscellaneous Statements

Splitting Records into Individual Fields

The `split` command breaks out regular expression delimited fields from an input string and places these fields into a list of strings:

```
($a,$b,...,$n) = split(/regex/,$string);
```

For example:

```
@fields = split(/:/,$passwd_record);
```

Joining Fields into Records

Opposite of the `split` command. Used as follows:

```
$record = join($delimiter,@list);
```

Removing the Last Character of a String

The `chop` command takes a string and chops off the last character:

```
chop($string);
```

This is particularly useful to remove the newline character from a line of input.

```
while ($record = <DATA>) {
    chop($record);
    process $record
}
```

8. Subroutines

Subroutines, with arguments and return values, are used to make Perl programs more modular. Subroutines, which can be placed anywhere in the program, are defined as follows:

```
sub subroutine-name {
    statement-block
}
```

The last expression evaluated in a subroutine is by default it's return value. Other values (either scalars or arrays) can be returned using an explicit return statement with a specified value:

```
return value;
```

By default, all variables are global to the entire program. Local variables can be declared using the `local` statement:

```
local (variable-name);
```

All variables not instantiated with `local` are global to the entire program.

Subroutines are invoked as follows:

```
&subroutine-name(argument-list);
```

Subroutine arguments are assigned to the '@_' variable which is local to the subroutine.

The following example simply multiplies two numbers and returns the result:

```
$a = 3;
$b = 12;
$c = &mult($a,$b);
    .
    .
    .

# Define multiply subroutine

sub mult {
    local($first,$second) = @_;
    local($result);

    $result = $first * $second;

    return $result;
}
```

Example 2 Multiply Subroutine

9. Perl Availability and Installation

Perl, distributed under the GNU public license, is available via anonymous FTP from several locations. Perl can be run on many types of systems, including:

- UNIX
- Macintosh
- DOS
- VMS

Perl is also available on Code 5595's Andrew File System (AFS) server as part of the online software repository and can be accessed from any UNIX system at the Lab. (Contact Computational Support Services at 767-3884 for more information about AFS and installing an AFS client on your system.)

These notes were developed for Perl 4.0, patch level 36 (Perl 4.036). Perl 5.0 is also available and has many additional features including object oriented programming. Everything covered in these notes should also work under Perl 5.

10. References

Many online and hardcopy references were used in creating this class. Among the best of these are the following:

- "Learning Perl, 2nd Edition" (aka the "Llama Book")

  by Randal L. Schwartz & Tom Christiansen
  O'Reilly and Associates, Inc.
  ISBN: 1-56592-284-0

- "Programming Perl, 2nd Edition" (aka the "Camel Book")

  by Larry Wall, Tom Christiansen & Randal L. Schwartz
  O'Reilly and Associates, Inc.
  ISBN: 1-56592-149-6

- "Advanced Perl Programming"

  by Sriram Srinivasan
  O'Reilly and Associates, Inc.
  ISBN: 1-56592-220-4

- the "Perl reference materials" Web page at:

  `http://www.eecs.nwu.edu/perl/perl.html`

- the newsgroup `comp.lang.perl`

11. Summary

Perl has become the "tool of choice" for many system administrators and Web page designers, as it allows you to do everything you can do using shell scripts, the `sed` and `awk` utilities, the C programming language, and many other commands as well.

Hopefully this class will put you well on the way to becoming an effective Perl programmer.